# Configuración Eco-Eficiente de Atributos de Calidad Funcionales

José Miguel Horcas, Mónica Pinto y Lidia Fuentes

Universidad de Málaga, Andalucía Tech, Spain
{horcas,pinto,lff}@lcc.uma.es, http://caosd.lcc.uma.es/

**Resumen** Los atributos de calidad funcionales (FQAs) son aquellos que para satisfacerlos se necesita incorporar funcionalidad adicional a la arquitectura de las aplicación (e.g., seguridad). La nueva funcionalidad incorporada por estos FQAs (e.g., encriptación) afecta a otro atributo de calidad como es el consumo de energía de la aplicación. Hasta el momento no se han explorado suficientemente las interdependencias entre, por ejemplo diferentes niveles de seguridad y su incidencia en el consumo de energía. En este artículo se propone una solución para ayudar al arquitecto software a generar la configuración de los FQAs que optimiza la eficiencia energética de la aplicación. Para ello se define un modelo de uso para cada FQA, teniendo en cuenta las variables que influyen en el consumo de energía y como el valor de estas variables cambia en función del punto de la aplicación donde se requiere ese FQA. Se extiende una Línea de Productos Software que modela una familia de FQAs para incorporar la variabilidad del modelo de uso y los frameworks existentes que implementan los FQAs. Generamos la configuración más eco-eficiente seleccionando el framework y las características más adecuadas para cada FQA y configurándolo según los requisitos de la aplicación.

**Keywords:** Atributos de Calidad, Eco-Eficiente, Energía, FQA, Línea de Productos Software, Variabilidad

## 1. Introducción

Los atributos de calidad o propiedades no funcionales de una aplicación (e.g., seguridad, usabilidad,...) pueden mejorarse modificando apropiadamente su arquitectura software con funcionalidad específica que ayude a satisfacer dicho atributo de calidad. Por ejemplo, el rendimiento de una aplicación web puede mejorarse añadiendo un componente de caché que sirva los datos almacenados de forma más rápida en futuras peticiones al servidor. Estas funcionalidades adicionales (e.g., el sistema de caché) que mejoran los atributos de calidad de la aplicación son conocidos como Atributos de Calidad Funcionales (FQAs, de *Functional Quality Attributes*) [6,9,17], ya que implican añadir funcionalidad adicional a la arquitectura de la aplicación que no está definida explícitamente en los requisitos.

Una característica esencial de los FQAs es que son recurrentes, lo que significa que son requeridos por diferentes aplicaciones para satisfacer sus requisitos de calidad. Por ejemplo, una aplicación puede requerir integridad y encriptación para garantizar la seguridad de los datos, mientras que otra sólo necesita encriptación, y/o el algoritmo de encriptación que necesita es distinto. Existen

multitud de bibliotecas y frameworks que proporcionan diferentes implementaciones de los FQAs listos para ser reutilizados en las aplicaciones, como por ejemplo el paquete Java Security, la biblioteca de Apache Commons, el framework Spring, etc. Sin embargo, no todas ellas satisfacen los requisitos de calidad de las aplicaciones de la misma forma. Diferentes implementaciones del mismo FQA ofrecen diferentes niveles de calidad de servicio (e.g., más o menos seguridad o usabilidad), pero normalmente a costa de hacer un mayor o menor uso de recursos. El uso de recursos normalmente incide en otro atributo de calidad que recientemente está cobrando mucha importancia, como es la eficiencia energética. Hasta el momento no se ha estudiado lo suficiente las interdependencias entre los FQAs y su consumo de energía, a pesar de que la eficiencia energética es cada vez más un requisito deseable a la hora de desarrollar aplicaciones.

En este artículo nos centraremos en mejorar la eficiencia energética de los FQAs de las aplicaciones por lo que el objetivo es generar aquellas configuraciones de los FQAs que sean más eco-eficientes y que a la vez se adapten a las necesidades de la aplicación — entendemos por configuración eco-eficiente aquella que consume menos energía que otras. En un trabajo previo definimos una familia de FQAs y un proceso basado en Línea de Productos Software (SPL, de *Software Product Line*) [16] que genera e inyecta configuraciones a medida de FQAs, en la arquitectura de las aplicaciones [9] (WeaFQAs). Sin embargo, no basta con configurar los diferentes componentes de los FQAs para una determinada aplicación, sino que habría que estudiar qué relación hay entre las diferentes variantes de los FQAs y otros atributos de calidad no funcionales, en este caso, la eficiencia energética. Si tenemos en cuenta que el arquitecto del software normalmente no es consciente del gasto energético que implican sus soluciones arquitectónicas, se hace necesario automatizar el proceso de generación de configuraciones de FQAs teniendo en cuenta el consumo de energía de cada una de las variantes. Para ello en primer lugar hay que tener en cuenta que un mismo FQA puede ser incorporado en diferentes puntos de la aplicación. Por ejemplo, el sistema de caché puede ser añadido en aquellos puntos de la arquitectura software donde el acceso a los datos suponga un cuello de botella para el rendimiento de la aplicación. Pero incorporar el mismo FQAs en diferentes partes de la aplicación implica que el FQA debe ser configurado de forma diferente dependiendo de las características de la interacción entre los componentes donde se agregue. En nuestro ejemplo, la aplicación requerirá diferentes caches en función del tipo y tamaño de los datos gestionados en cada interacción. Esto implica tener que configurar diferentes instancias de un FQA, teniendo en cuenta que se busca optimizar el consumo energético global de la aplicación y que distintos tipos de caché y distintas implementaciones de las mismas tendrán distinto consumo energético, que dependerá además del patrón de uso. Para ello, definiremos un modelo de uso para cada FQA teniendo en cuenta las variables que influyen en el consumo de energía (e.g., tamaño del mensaje a cifrar en el caso de la encriptación, frecuencia de accesp a los datos para el FQA de persistencia, etc.) en función del punto de la aplicación donde se requiere ese FQA.

Por lo tanto, en este artículo proponemos una solución para ayudar al arquitecto software a generar la configuración de los FQAs que optimiza la eficiencia

energética global de la aplicación, extendiendo la propuesta WeaFQAs definida en [9] para: (1) definir cada FQA como una característica clonable que permita generar diferentes configuraciones de un mismo FQA; (2) añadir el modelo de uso de cada FQA al modelo de variabilidad de forma que el arquitecto software pueda instanciarlo para cada punto de la aplicación donde se vaya a incorporar el FQA; (3) añadir los diferentes frameworks y bibliotecas reales que implementan los FQAs haciendo explícita las configuraciones que ofrecen, y (4) generar la configuración más eco-eficiente seleccionando el framework más adecuado para cada FQA y configurándolo a los requisitos de la aplicación. Implementamos nuestra propuesta usando el lenguaje CVL (*Common Variability Language*) [8] y medimos el consumo de energía de los diferentes frameworks en Java.

El resto del artículo se estructura de la siguiente manera. La Sección 2 muestra los antecedentes y el trabajo relacionado. En la Sección 3 se define el modelo de uso para cada FQA. La Sección 4 extiende nuestro trabajo previo con los modelos de uso y los frameworks de los FQAs. La Sección 5 detalla cómo se genera una configuración a partir de los resultados de la experimentación. La Sección 6 evalúa y discute nuestra propuesta. La Sección 7 concluye el artículo.

## 2. Trabajo Relacionado y Antecedentes

En esta sección se discuten las propuestas que existen para modelar los FQAs e incorporarlos a las aplicaciones, haciendo hincapié en la propuesta WeaFQAs [9] y sus limitaciones. También se presenta el trabajo relacionado sobre eco-eficiencia a nivel de arquitectura software.

### 2.1. Atributos de Calidad Funcionales (FQAs)

Existen distintos enfoques para trabajar con los FQAs. En [6], los autores defienden que la mayoría de los requisitos no funcionales realmente describen propiedades del comportamiento del sistema y que, por tanto, deberían tratarse de la misma forma que los requisitos funcionales. Llegan a esta conclusión tras analizar y clasificar más de 500 requisitos no funcionales de distintas especificaciones de requisitos de la industria. En [17], los autores aplican un método inductivo de investigación para identificar FQAs. En concreto, identifican funcionalidades del atributo de usabilidad que afectan a la arquitectura software, y definen patrones para implementar esas funcionalidades en diferentes aplicaciones. En [2,18], los autores presentan el paradigma de desarrollo software CORE (de *Concern-Oriented Reuse*). En CORE, todo tipo de características software, desde la funcionalidad base, incluyendo los FQAs, hasta las propiedades no funcionales, son encapsuladas en unidades reutilizables llamadas *concerns*. Aunque no trabajan explícitamente con el concepto de FQA, la funcionalidad necesaria para satisfacer estos atributos funcionales la identifican y encapsulan en los *concerns*. Por último, en [9] se define WeaFQAs, una propuesta para modelar una familia completa de FQAs separadamente de la funcionalidad principal de la aplicación, siguiendo un enfoque de Línea de Productos Software (SPL). Combinando el enfoque de SPL con orientación a aspectos se define un proceso genérico para modelar, configurar, e incorporar automáticamente los FQAs a la arquitectura software de las aplicaciones.

Como el objetivo de este artículo es extender WeaFQAs, a continuación se discuten las similitudes y diferencias del resto de propuestas con WeaFQAs. WeaFQAs comparte la definición de FQAs con [6], y la necesidad de tratar estos atributos como cualquier otro requisito funcional. También comparte con [17] la idea de identificar y modelar las funcionalidades asociadas a un atributo de calidad, aunque mientras [17] se centra en la usabilidad, WeaFQAs es una propuesta más genérica que da soporte para cualquier FQA. De hecho, las funcionalidades de usabilidad identificadas en [17] podrían incorporarse a WeaFQAs para enriquecer su modelo de varialidad. Además, WeaFQAs modela e incorpora los FQAs de forma menos intrusiva a la arquitectura software de las aplicaciones, gracias a la orientación a aspectos. Por último, con respecto a CORE [2,18], ellos modelan la variabilidad de los *concerns* a nivel de su interfaz, en lugar de modelar la variabilidad de la funcionalidad interna de los componentes como hace WeaFQAs [9] para los FQAs, por lo que la generación de configuraciones en CORE está limitada a variaciones de alto nivel. Ninguna de estas propuestas, incluyendo WeaFQAs, está preparada para poder realizar un análisis del consumo de energía de distintas arquitecturas software alternativas. Esto es debido principalmente a que no incluyen soporte para representar el modelo de uso de los FQAs, ni tampoco incluyen las características configurables de los frameworks e implementaciones reales de los FQAs, algo que es imprescindible si se quiere analizar el consumo de energía de una configuración arquitectónica determinada.

### 2.2. Eficiencia energética

Recientemente, la eficiencia energética está siendo considerada como un nuevo atributo de calidad de los sistemas software [10], que aunque es claramente un atributo de calidad no funcional, se ve afectado por la funcionalidad del sistema y en nuestro caso por los FQAs [7,11]. Por ejemplo, la sostenibilidad se propone como un atributo de calidad en [10], que se centra en el consumo de recursos y se descompone en las siguiente características: utilización del software, uso de la energía, y carga de trabajo, que afectan directamente a la eficiencia energética. En muchas aplicaciones el consumo de energía es un atributo de calidad crítico, como por ejemplo en sistemas de redes *mesh* para la Internet de las Cosas [1,5].

La importancia de razonar a nivel arquitectónico sobre eficiencia energética es poder comparar el consumo de energía de diferentes configuraciones arquitectónicas de una misma aplicación (patrones, alternativas de diseño, despliegue, etc.). Existen varias propuestas enfocadas en la definición de tácticas arquitectónicas [10] y diseño de patrones [13] guiados por la energía, así como nuevos lenguajes de descripción de arquitecturas que incorporan perfiles de energía y análisis del consumo de energía [14,20]. La parte experimental en cualquier caso consiste en medir el consumo de energía de la aplicación a nivel de código para analizar los efectos de aplicar un patrón arquitectónico o diseño específico [10,13].

Con respecto a la configuración y optimización de configuraciones basadas en atributos de calidad, existen múltiples trabajos [12,19,21] aunque la mayoría se centran en realización de pruebas la estimación de los atributos de calidad, y ninguno considera la eficiencia energética como un atributo de calidad.

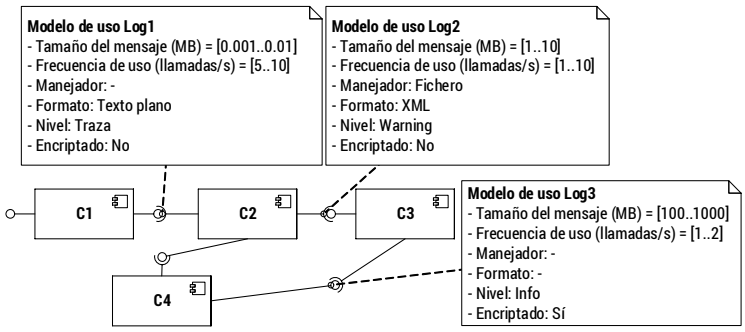


**Figura 1.** Arquitectura de ejemplo con modelos de uso para el *logging*.

## 3. FQAs y Eficiencia Energética

Para poder analizar y seleccionar aquellas configuraciones de los FQAs que son más eficientes energéticamente primero debemos caracterizar cómo estos afectan al consumo de energía. Esto significa identificar el conjunto de variables propias de la funcionalidad ofrecida por el FQA que influyen en el consumo de energía. Para cada funcionalidad de los FQAs se identifican las variables que afectan al consumo energético, así como algunos valores de ejemplo que pueden tomar esas variables. Por ejemplo, la funcionalidad de *logging* del FQA de usabilidad tiene mucha variabilidad puesto que los mensajes a registrar pueden variar de formato (e.g., texto plano, XML,...), se pueden enviar a diferentes salidas (manejadores) como consola o fichero, pueden tener diferentes niveles de severidad (e.g., traza, debug,...), o pueden encriptarse si fuera necesario.

El efecto de una de estas variables en el consumo de energía de un FQA se conoce a través de la experimentación, pues hay que evaluar su consumo para los diferentes valores (llamados *treatments* en experimentación) que pueda tomar dicha variable (factor) [22]. Como ejemplo de esta experimentación, en la Sección 5.1 se muestran los resultados de evaluar el consumo de energía de la funcionalidad de *logging* para diferentes frameworks Java que la implementan. Nótese que los resultados concretos de consumos no son importantes en la caracterización del FQA, sino el comportamiento energético del FQA cuando varían los valores que puedan tomar sus variables. Por este motivo en este artículo asumimos que la caracterización y análisis del consumo energético puede realizarse de manera independiente a una aplicación concreta. En cualquier caso, como discutimos en la evaluación, esto habrá que confirmarlo replicando la experimentación con aplicaciones reales que incluyan dichos FQAs.

La forma en la que el consumo energético de un FQA afecta en una aplicación concreta viene determinada por el modelo de uso del FQA en las diferentes partes de la aplicación. Se define *modelo de uso* de un FQA como el conjunto de variables que influyen en el consumo de energía de un FQA y los valores que cada variable puede tomar (e.g., el tamaño de los mensajes que se generan estará siempre entre los 10 y los 100 KBytes) [3,20]. La caracterización de los FQAs nos permite construir un modelo de uso genérico que puede ser instanciado múltiples veces en una aplicación por el arquitecto software, en diferentes puntos de la aplicación. La Figura 1 muestra tres modelos de uso diferentes de la funcionalidad *logging*

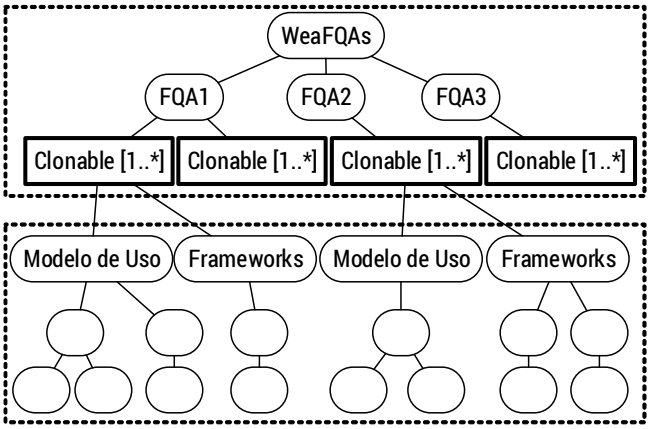


**Figura 2.** Modelo de variabilidad con modelos de uso y frameworks.

en tres puntos de una arquitectura software donde se pretende incorporar esa funcionalidad. Los valores del modelo de uso pueden ser difíciles de asignar o pueden no ser conocidos por el arquitecto software. O simplemente el arquitecto no quiere proporcionarlos para poder realizar un análisis del consumo de energía más amplio. En estos casos, nuestra propuesta permite instanciar parcialmente el modelo de uso como se muestra en el `Modelo de uso Log1` y el `Modelo de uso Log3` del ejemplo de la Figura 1, donde no se ha proporcionado la información del manejador de *logging* (donde se enviará el *logging*: consola o fichero) en el primer caso, y tampoco se ha proporcionado el formato (texto plano, XML) en el segundo caso.

En la siguiente sección se detalla como se extiende el modelo de variabilidad de WeaFQAs para incluir, entre otras cosas, el modelo de uso de los FQAs.

## 4. Extensión del Modelo de Variabilidad de WeaFQAs

En esta sección presentamos todas las modificaciones a la SPL de la familia de FQAs definida en [9] (WeaFQAs) que se comentaron en la Sección 2. La Figura 2 muestra un esquema del modelo de variabilidad extendido.

Por un lado, el modelo de variabilidad especifica una familia completa de FQAs (e.g., security, usability,...) y las funcionalidades de cada uno (e.g., encriptación, *logging*,...). Para poder generar configuraciones distintas del mismo FQA para una misma aplicación, en este artículo hemos modificado las características del árbol de variabilidad que representan las funcionalidades de los FQAs [9] para hacerlas clonables (características con [1..*] junto al nombre). Una característica clonable del árbol de variabilidad permite instanciar dicha característica múltiples veces, y todas sus sub-características pueden ser configuradas de forma diferente para cada instancia. La cardinalidad [1..*] de la característica clonable indica el número de instancias que se pueden generar.

Por otro lado, para permitir al arquitecto software instanciar los modelos de uso y generar las configuraciones más eficientes de los FQAs, proponemos extender el modelo de variabilidad de los FQAs definido en [9]. Por un lado, hemos ampliado el modelo de variabilidad incluyendo, para cada funcionalidad

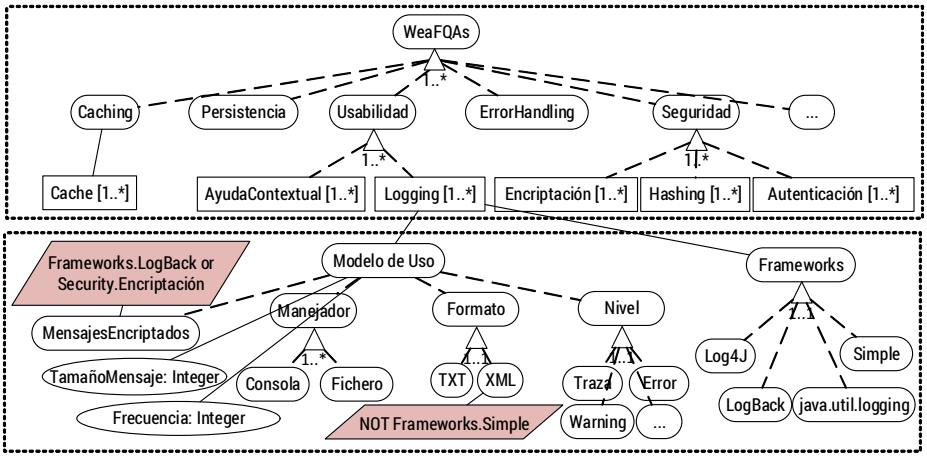


**Figura 3.** Modelo de variabilidad extendido para *logging*.

de los FQAs, todas las variables propias de ese FQA que influyen en el consumo energético (i.e., el modelo de uso). Por otro lado, hemos incluido también los frameworks reales que la implementan (parte inferior de la Figura 2). Esto permite seleccionar para cada funcionalidad el framework más adecuado en función del modelo de uso de dicha funcionalidad. Hacer clonable la característica del FQA también implica que se pueden generar múltiples instancias del modelo de uso y seleccionar un framework para cada instancia. Nótese que puede haber variables del modelo de uso de un FQA que no todos los frameworks reales proporcionan, en estos casos se tendrían que definir restricciones en el árbol de características para restringir las posibles selecciones. Cabe destacar el uso de CVL (Common Variability Language) [8] para modelar la variabilidad debido a que permite especificar atributos con valores para las variables del modelo de uso, además de definir características clonables sin necesidad de extender su meta-modelo como ocurre con los modelos de características tradicionales (*feature models*) [4].

La Figura 3 muestra un ejemplo del modelo de variabilidad de WeaFQAs extendido con el modelo de uso y las características de los frameworks para la funcionalidad de *logging*. Primero, la funcionalidad se ha hecho clonable (`Logging [1..*]`) para permitir múltiples instancias de dicha funcionalidad. Luego, se especifican todas las variables del modelo de uso (e.g., tamaño del mensaje, formato, manejador, etc.) y algunos de los frameworks Java que implementan la funcionalidad de *logging*: `Log4J`, `LogBack`, la biblioteca `java.util.logging`, y una implementación básica de la API SLF4J (`simple`). Nótese cómo el único framework que proporciona la funcionalidad de encriptar los mensajes es `LogBack`, y por lo tanto hemos definido una restricción asociada a esa característica que impide seleccionar otro framework diferente en caso de querer encriptar los mensajes a registrar (`MensajesEncriptados implies Framework.LogBack or Security.Encriptación`). En este caso, el arquitecto software estará obligado a escoger el framework `LogBack` o proporcionar la encriptación de forma independente al framework de logging, a través del FQA de seguridad. Igualmente, el único framework que no permite registrar los mensajes en formato XML

es la implementación simple de SLF4J, por lo que se especifica la restricción `Formato.XML implies NOT Frameworks.Simple`.

## 5. Generación de una Configuración Eco-Eficiente

Usando el modelo de variabilidad, el arquitecto software puede generar una configuración total o parcial de los FQAs de acuerdo a los requisitos de la aplicación. El objetivo aquí es ayudar al arquitecto a generar aquella configuración más eco-eficiente. Para ello, en primer lugar es necesario evaluar el efecto de las diferentes variables del modelo de uso en el consumo de energía independientemente de la aplicación con la que luego se vaya a componer el FQA.

### 5.1. Experimentación: consumo energético

Para ilustrar los detalles de nuestra propuesta por medio de un ejemplo, en esta sección mostramos los resultados de haber evaluado el consumo de energía de la funcionalidad de *logging* del FQA de usabilidad para diferentes frameworks Java según la caracterización realizada en la Sección 3.

Los experimentos se realizaron en un ordenador Intel Core i7-4770, 3.40 GHz, 16 GB de memoria, sistema operativo Windows 10 de 64 bits y Java JDK 1.8. Las mediciones de energía se han realizado con ayuda de la herramienta IPPET (*Intel Platform Power Estimation Tool*)[1] que monitoriza la energía consumida por la CPU a nivel de proceso. Los experimentos para caracterizar la funcionalidad de *logging* consistieron en registrar una serie de mensajes de log usando cuatro implementaciones diferentes de la API de logging SLF4J (*Simple Logging Facade for Java*)[2]: el paquete java.util.logging proporcionado directamente por el JDK de Java, los frameworks Log4J[3] y LogBack[4], y una implementación simple de SLF4J[5]. Se registraron mensajes de texto plano variando el tamaño desde 100 Bytes a 1 GByte en potencias de 10. Cada variable o característica del modelo de uso ha sido evaluada de forma independiente, fijando el resto de variables. Cada prueba se realizó cinco veces obteniendo la mediana de las pruebas.

Las Figuras 4 y 5 muestran los resultados de las mediciones de energía (en Julios) del *logging* para los diferentes tamaños de mensaje, usando dos manejadores diferentes: envío a consola y fichero. Por un lado se observa que el framework más eco-eficiente en general es la implementación simple de SLF4J, mientras que el framework LogBack es el que más energía consume en todos los casos (excepto para mensajes de 1 GB). Se observa también que para mensajes de gran tamaño (1 GB) es más eco-eficiente enviar el mensaje a fichero en lugar de mostrarlo por pantalla. Mensajes de log de semejante tamaño no son comunes en aplicaciones de escritorio o móviles, pero sí en aplicaciones de mayor escala como por ejemplo las granjas de servidores de Google[6] o los grandes centros de datos del CERN[7], donde se realizan registros del estado completo de las aplicaciones con mucha

[1] https://software.intel.com/es-es/forums/software-tuning-performance-optimization-platform-monitoring/topic/518161
[2] https://www.slf4j.org/
[3] http://logging.apache.org/log4j/1.2/index.html
[4] https://logback.qos.ch/
[5] https://www.slf4j.org/apidocs/org/slf4j/impl/SimpleLogger.html
[6] https://www.google.com/about/datacenters/
[7] http://information-technology.web.cern.ch/about/computer-centre

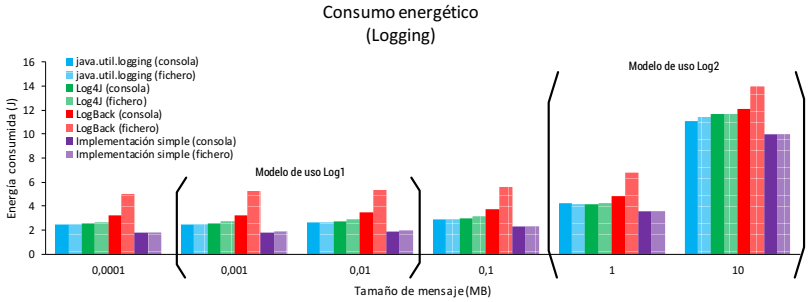


**Figura 4.** Consumo de energía del *logging* para mensajes entre 100 B y 10 MB.

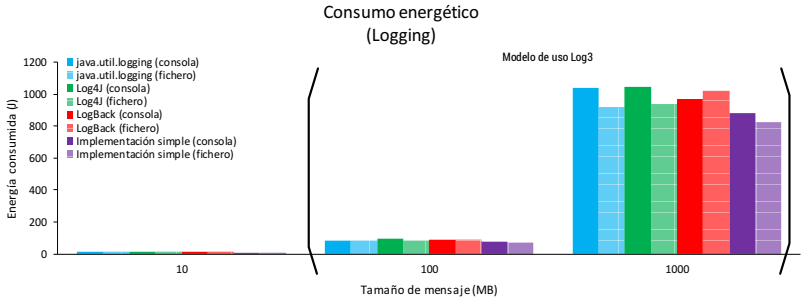


**Figura 5.** Consumo de energía del *logging* para mensajes entre 10 MB y 1 GB.

frecuencia, y donde la eficiencia energética es muy importante. El mayor consumo observado en los frameworks Log4J y LogBack se debe a que son frameworks de logging muy completos que proporcionan mucha funcionalidad y se enfocan en proporcionar un gran rendimiento sin tener en cuenta el consumo de energía.

### 5.2. Configuración eco-eficiente

A partir de los resultados obtenidos de la experimentación, nuestra propuesta es capaz de seleccionar la configuración de los FQA más eco-eficiente en función de los requisitos de la aplicación. Para cada FQA que requiera la aplicación, el arquitecto software debe primero identificar los puntos (o interacciones) de la arquitectura de la aplicación donde va a incorporar el FQA. Segundo, para cada punto debe crear una instancia del modelo de uso de ese FQA. Esto implica que debe proporcionar los valores específicos de las variables que componen el modelo de uso. Es decir, debe especificar cómo va a usarse ese FQA en ese punto particular de la aplicación.

Si el arquitecto genera una configuración total para un FQA (es decir, proporciona todos los valores del modelo de uso y selecciona un framework en particular, porque los requisitos de la aplicación así lo requieren), la configuración obtenida podría no ser la más eco-eficiente. Sin embargo, cuando el arquitecto instancia parcialmente los modelos de uso, nuestra propuesta completa automáticamente la configuración más eco-eficiente para las características y valores dados, seleccionando aquellas características restantes que minimizan el consumo, como por ejemplo el framework con menor consumo según cada modelo de uso. Además, el arquitecto podría exigir usar un determinado framework, pero podría

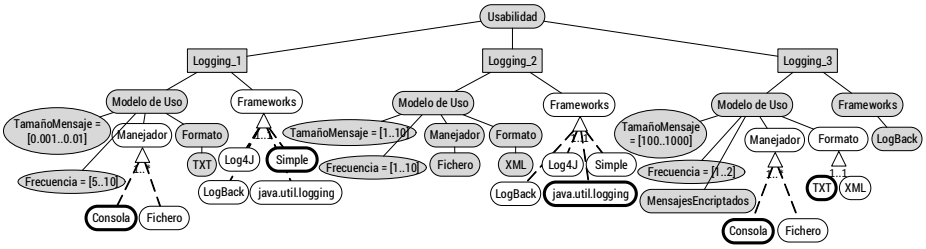


**Figura 6.** Configuración de tres instancias de *logging*.

no configurar sus características. En estos casos nuestra propuesta generará la configuración más eco-eficiente teniendo en cuenta las características y variables de ese framework en particular.

La selección de la configuración más eco-eficiente se realiza minimizando la Ecuación 1. El consumo energético de una configuración global de los FQAs es la suma de las diferentes configuraciones (`config`) de cada instancia ($i$) de las funcionalidades ($f$) de los FQAs. El consumo de energía de cada instancia se calcula a partir del modelo de uso del FQAs, usando los valores obtenidos mediante experimentación (ver Sección 5.1). La solución de la función objetivo puede obtenerse por medio de cualquier técnica de optimización, como por ejemplo programación con restricciones o algoritmos evolutivos [15], aunque en este artículo no detallamos una propuesta concreta.

$$\texttt{Consumo\_energético}(FQAs) = \sum_{f \in FQAs} \sum_{i=1}^{N} \texttt{Consumo\_energía}(config_i(f)) \quad (1)$$

La Figura 6 muestra una configuración del modelo de variabilidad con las tres instancias de la funcionalidad de *logging* configuradas con los valores de los modelos de uso mostrados en la Figura 1 de la Sección 3. En la primera interacción (`Modelo de uso Log1`), el arquitecto software ha definido un tamaño de mensaje de entre 0.001 MB y 0.01 MB, y una frecuencia de uso de entre 5 y 10 invocaciones por segundo. Los mensajes se mostrarán a nivel de traza en texto plano y sin encriptar. Sin embargo, el arquitecto no ha especificado a qué manejador será enviado el mensaje (consola o fichero) ni el framework a usar, por lo que nuestra propuesta seleccionará automáticamente el manejador más eco-eficiente (i.e., consola) para los valores proporcionados, así como el framework de logging más eco-eficiente que proporciona las características seleccionadas: la implementación simple de SLF4J. En el `Modelo de uso Log2`, el arquitecto ha proporcionado una configuración total, pero no ha seleccionado el framework a usar. En este caso nuestra propuesta seleccionará la biblioteca java.util.logging debido a que es la opción más eco-eficiente que cumple las restricciones del modelo de variabilidad, es decir, es el más eco-eficiente que permite registrar mensajes entre 1 y 10 MB a fichero en formato XML (véase la restricción en la Figura 3). Por último, el arquitecto ha seleccionado una configuración parcial para el `Modelo de uso Log3`, seleccionando la opción de encriptar los mensajes y por consecuencia el framework LogBack, que es el único que proporciona esa funcionalidad. Nuestra propuesta seleccionará el manejador de consola y el formato de texto plano (`TXT`) que es la configuración más eco-eficiente para mensajes de entre 100 MB y 1 GB cuando se usa el framework LogBack.

## 6. Evaluación

En esta sección evaluamos nuestra propuesta demostrando que las configuraciones generadas son correctas y son las más eco-eficientes para cada modelo de uso. Para ello, representamos el árbol de variabilidad y las configuraciones parciales de los FQAs como un problema de satisfacción de restricciones (CSP, de *Constraint Satisfaction Problem*). Formalmente, un problema CSP está definido por una tripleta $(X, D, C)$, dónde $X$ es el conjunto de variables, $D$ es el dominio de las variables, y $C$ es el conjunto de restricciones que se deben satisfacer. Extendemos esta definición para incluir la información de energía necesaria para generar las configuraciones más eco-eficientes:

- **Variables ($X$):** $X = V_S \cup V_V \cup E$,
  donde $V_S$ es el conjunto de variables que representan todas las características del árbol; $V_V$ es el conjunto de variables que representan aquellas características que pueden tomar valores específicos como por ejemplo el tamaño del mensaje del *logging*; y $E$ es el conjunto de variables que representan la energía consumida por cada posible configuración completa de un FQA. Por ejemplo, para el *logging* tendríamos los siguientes conjuntos de variables:
  $V_{S_{logging}} = \{x_{tamañomensaje}, x_{consola}, x_{fichero}, x_{txt}, x_{xml}, x_{logback}, x_{log4j}, ...\}$
  $V_{V_{logging}} = \{v_{tamañomensaje}, v_{frecuencia}\}$
  $E_{logging} = \{e_{configLog1}, e_{configLog2}, e_{configLog3}, ...\}$
- **Dominio ($D$):** $\forall v \in V_S, v \in \{0,1\}$, $\forall v \in V_V, v \in Dom(v)$, y $\forall e \in E, e \in \mathbb{R}$,
  es decir, las variables que representan las características del árbol ($V_S$) puede tomar los valores $\{0,1\}$ según se seleccione o no en una configuración; las variables que representan características que pueden tomar un valor de un tipo específico ($V_V$) tienen el dominio definido en el propio árbol de variabilidad, por ejemplo el dominio de la variable $v_{frecuencia}$ es el conjunto de los números naturales ($\mathbb{N}$); mientras que las variables que representan la energía consumida por cada configuración toman valores reales (en Julios).
- **Restricciones ($C$):** $C = C_R \cup C_E \cup C_C$,
  donde $C_R$ es el conjunto de restricciones que representan las relaciones del árbol de variabilidad y las restricciones entre ramas. Un ejemplo de este tipo de restricción es $x_{log4j} + x_{logback} + x_{javautillogging} + x_{simple} = 1$, para asegurar que sólo un framework de *logging* es seleccionado en una configuración particular. Otra restricción en $C_R$ es $x_{modelodeuso} = x_{tamañomensaje}$, que modela la relación obligatoria padre-hijo entre las características de `Modelo de Uso` y `TamañoMensaje` del árbol de variabilidad (Figura 3). $C_E$ es el conjunto de restricciones que definen los valores de energía de una configuración particular obtenidos a partir de la experimentación (ver Sección 5.1). Un ejemplo de este tipo de restricción es: $x_{consola} \wedge x_{txt} \wedge x_{log4j} \wedge x_{tamañomensaje} \wedge v_{tamañomensaje} \geq 0{,}01 \wedge v_{tamañomensaje} \leq 10 \implies e_{configLog1}$ que junto a la restricción $e_{configLog1} = 11{,}6736$ representan un modelo de uso instanciado completamente. Para simplificar el problema CSP se ha escogido el peor caso de consumo energético para el intervalo de valores dado en el tamaño de mensaje. Por último, $C_C$ es el conjunto de restricciones que modelan las selecciones realizadas por el arquitecto software en una configuración del modelo de variabilidad. Por ejemplo, las siguientes restricciones modelan la

configuración parcial de la instancia `Logging_1` de la Figura 6:
(1) $x_{modelodeuso1} = 1$, (2) $x_{tamañomensaje1} = 1$, (3) $x_{frecuencia1} = 1$, (4) $x_{formato1} = 1$, (5) $x_{txt1} = 1$, (6) $v_{tamañomensaje1} \geq 0{,}001 \wedge v_{tamañomensaje1} \leq 0{,}01$, y (7) $v_{frecuencia1} \geq 5 \wedge v_{frecuencia1} \leq 10$.
El sufijo '1' tras el nombre de las variables identifica unívocamente las diferentes instancias de la característica clonables de *logging*. Las restricciones $C_R$ y $C_E$ son aplicables a todas las instancias de un FQA, mientras que la restricciones $C_C$ son propias de la configuración de cada instancia.

Definimos la función objetivo del problema CSP que genera las configuraciones más eco-eficientes sujeta a las restricciones definidas (Ecuación 2). Para resolver el problema CSP, en este artículo, hemos usado Choco[8], una biblioteca Java de código abierto para programación de restricciones. Las configuraciones generadas con Choco $conf_i = x_1, x_2, ...x_n$ cumplen con el modelo de uso introducido por el arquitecto software, garantizando que las configuraciones generadas por nuestra propuesta son las más eco-eficientes.

$$\texttt{Minimizar} \sum_{i=1}^{|E|} e_i, \quad e_i \in E \tag{2}$$

### 6.1. Discusión

En esta sección discutimos algunas lecciones aprendidas durante el desarrollo y la evaluación de nuestra propuesta.

- **Selección de la configuración más eco-eficiente.** Gracias a las características clonables, nuestra propuesta permite generar múltiples instancias configuradas de manera diferente para la misma aplicación. Sin embargo, hay frameworks, como los frameworks de *logging* usados en este artículo, que no permiten usar más de un framework diferente en la misma aplicación si se está usando bajo la API de SLF4J. Este caso concreto puede solucionarse utilizando cada framework de manera independiente en cada instancia, sin hacer uso de la API SLF4J. Para un caso más general, nuestra propuesta puede extenderse para realizar un *trade-off* entre las diferentes instancias del modelo de uso y seleccionar el framework más eco-eficiente valorando todas las instancias en común. En el ejemplo propuesto en este artículo, si los componentes de la aplicación están distribuidos no hay problema en usar un framework de *logging* diferente en diferentes máquinas virtuales de Java.
- **Evaluación de consumo de energía de FQAs en aplicaciones reales.** El consumo real de las configuraciones generadas de los FQAs puede variar debido a factores independientes del modelo de uso como por ejemplo la carga del sistema en un momento determinado, o el grado de compartición de recursos (e.g., uso de red, disco). El efecto de éstos factores en el consumo de energía de los FQAs no se conoce a priori, pero podrían modelarse explícitamente para tenerlos en cuenta en la propuesta de manera independiente del modelo de uso, y realizar los experimentos según los diferentes estados o contextos en los que se encuentre la aplicación.

[8] `http://www.choco-solver.org/`

- ***Trade-off* entre atributos de calidad.** Aunque en este artículo nos hemos centrado en la eficiencia energética, los FQAs también afectan a otras propiedades no funcionales como el rendimiento y el nivel de seguridad o usabilidad. Nuestra propuesta puede usarse para generar aquellas configuraciones que optimicen otro atributo de calidad en lugar de la eficiencia energética, caracterizando los FQAs con las variables que afecten a ese atributo de calidad y modificando el modelo de uso si fuera necesario. Más aún, la propuesta se puede extender para permitir realizar un *trade-off* entre varios atributos de calidad [10].
- **Determinismo de la propuesta.** Nuestra propuesta se basa en los datos obtenidos de la experimentación para generar la configuración más eco-eficiente. Realizando los mismos experimentos en otro entorno, los valores concretos de energía pueden variar debido a muchos factores (principalmente el hardware), pero los resultados comparativos deberían seguir siendo correctos. En casos concretos en los que el entorno de ejecución sea muy distinto al de una máquina de propósito general como es el caso de los super computadores u ordenadores desarrollados específicamente para resolver un determinado problema, los resultados de la experimentación pueden ser muy diferentes y, por lo tanto, nuestra propuesta no generará las mismas configuraciones. Sin embargo, la propuesta seguirá generando aquellas configuraciones más eco-eficiente para ese entorno de ejecución específico.

## 7. Conclusiones y Trabajo Futuro

En este artículo se ha presentado una propuesta para generar configuraciones eco-eficientes de atributos de calidad funcionales (FQAs). La propuesta es útil porque ayuda al arquitecto software a incorporar la configuración más apropiada de los FQAs en cada punto de la aplicación donde se requieran, optimizando la eficiencia energética mediante la definición de un modelo de uso. Se ha evaluado el consumo energético de diferentes frameworks que implementan los FQAs.

Como trabajo futuro se pretende extender la propuesta para generar configuraciones de los FQAs haciendo un trade-off entre eficiencia energética, rendimiento, nivel de seguridad y nivel de usabilidad, entre otros atributos de calidad.

## Agradecimientos

Trabajo financiado por los proyectos MAGIC P12-TIC1814 y HADAS TIN2015-64841-R, y por la Universidad de Málaga.

## Referencias

1. Acosta Padilla, F.J.: Self-adaptation for Internet of things applications. Ph.D. thesis, Université Rennes 1 (2016)
2. Alam, O., Kienzle, J., Mussbacher, G.: Concern-oriented software design. In: Model-Driven Engineering Languages and Systems. pp. 604–621. MODELS (2013)
3. Becker, S., Koziolek, H., Reussner, R.: The palladio component model for model-driven performance prediction. Journal of Systems and Software 82(1), 3–22 (2009)
4. Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A.: Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In: International Conference on Software Engineering. pp. 472–481. ICSE (2013)

5. Dougherty, B., White, J., Schmidt, D.C.: Model-driven auto-scaling of green cloud computing infrastructure. Future Gener. Comput. Syst. 28(2), 371–378 (Feb 2012)
6. Eckhardt, J., Vogelsang, A., Fernández, D.M.: Are "non-functional" requirements really non-functional?: An investigation of non-functional requirements in practice. In: International Conference on Software Engineering. pp. 832–842. ICSE (2016)
7. Grosskop, K., Visser, J.: Identification of application-level energy optimizations. Proceeding of ICT for Sustainability (ICT4S) pp. 101–107 (2013)
8. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A.: Adding standardized variability to domain specific languages. In: International Software Product Line Conference. pp. 139–148. SPLC (2008)
9. Horcas, J.M., Pinto, M., Fuentes, L.: An automatic process for weaving functional quality attributes using a software product line approach. Journal of Systems and Software 112, 78–95 (2016)
10. Jagroep, E., van der Werf, J.M., Brinkkemper, S., Blom, L., van Vliet, R.: Extending software architecture views with an energy consumption perspective. Computing pp. 1–21 (2016)
11. Kalaitzoglou, G., Bruntink, M., Visser, J.: A practical model for evaluating the energy efficiency of software applications. In: ICT4S (2014)
12. Kolesnikov, S.S., Apel, S., Siegmund, N., Sobernig, S., Kästner, C., Senkaya, S.: Predicting quality attributes of software product lines using software and network measures and sampling. In: Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems. pp. 6:1–6:5. VaMoS '13, New York, NY, USA (2013)
13. Noureddine, A., Rajan, A.: Optimising energy consumption of design patterns. In: IEEE International Conference on Software Engineering. vol. 2, pp. 623–626 (2015)
14. Ouni, B., Rekhissa, H.B., Belleudy, C.: Inter-process communication energy estimation through aadl modeling. In: Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design. pp. 225–228. SMACD (2012)
15. Pascual, G.G., Pinto, M., Fuentes, L.: Self-adaptation of mobile systems driven by the common variability language. Future Generation Computer Systems 47, 127–144 (2015), special Section: Advanced Architectures for the Future Generation of Software-Intensive Systems
16. Pohl, K., Böckle, G., Linden, F.J.v.d.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc. (2005)
17. Rodríguez, F.D., Acuña, S.T., Juzgado, N.J.: Reusable solutions for implementing usability functionalities. International Journal of Software Engineering and Knowledge Engineering 25(4), 727–756 (2015)
18. Schöttle, M., Alam, O., Kienzle, J., Mussbacher, G.: On the modularization provided by concern-oriented reuse. In: Modularity. pp. 184–189 (2016)
19. Siegmund, N., RosenmüLler, M., KäStner, C., Giarrusso, P.G., Apel, S., Kolesnikov, S.S.: Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. Inf. Softw. Technol. 55(3), 491–507 (2013)
20. Stier, C., Koziolek, A., Groenda, H., Reussner, R.H.: Model-based energy efficiency analysis of software architectures. In: European Conference on Software Architecture. pp. 221–238. ECSA (2015)
21. Temple, P., Galindo Duarte, J.A., Acher, M., Jézéquel, J.M.: Using Machine Learning to Infer Constraints for Product Lines. In: Software Product Line Conference (SPLC) (2016)
22. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. Springer Science & Business Media (2012)